Lexical Analysis, I

Comp 412
Our goal is to automate the construction of scanners & parsers

**Scanner**
- Specify syntax with regular expressions (REs)
- Automatically build finite-automaton & scanner from the RE

**Parser**
- Specify syntax with context-free grammars (CFGs)
- Automatically build push-down automaton & parser from the CFG
In Lecture 2, we tried to define “positive integer”

- Is 001 a positive integer? What about 00?
- Automata are precise specifications; words are often ambiguous

Transition diagrams are precise. They are not concise.
We need a better notation to specify microsyntax.
Regular Expressions

We need a better notation to specify microsyntax

Regular Expressions over an Alphabet $\Sigma$

• If $x \in \Sigma$, then $x$ is an RE denoting the set $\{ x \}$ or the language $L = \{ x \}$
• If $x$ and $y$ are REs then
  - $xy$ is an RE denoting $L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \}$ (concatenation)
  - $x \mid y$ is an RE denoting $L(x) \cup L(y)$ (alternation)
  - $x^*$ is an RE denoting $L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k$ (Kleene Closure)
    $\rightarrow$ Set of all strings that are zero or more concatenations of $x$
  - $x^+$ is an RE denoting $L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k$ (Positive Closure)
    $\rightarrow$ Set of all strings that are one or more concatenations of $x$ (or $xx^*$)
• $\varepsilon$ is an RE denoting the empty set

Many RE-based systems support additional notation and operators. Those added features build on alternation, concatenation, and closure — plus, perhaps logical complement or negation. Complement is easy and efficient, if we think of the underlying DFA. (We will revisit this issue.)
Regular Expressions

Why these operators?

Regular Expressions over an Alphabet \( \Sigma \)

- If \( x \) is in \( \Sigma \), then \( x \) is an RE denoting the set \( \{ x \} \) or the language \( L = \{ x \} \)
  \( \rightarrow \) The spelling of any letter in the alphabet is an RE

- If \( x \) and \( y \) are REs then
  - \( xy \) is an RE denoting \( L(x)L(y) = \{ pq \mid p \in L(x) \text{ and } q \in L(y) \} \)
    \( \rightarrow \) If we concatenate letters, the result is an RE, so we can spell words
  - \( x \mid y \) is an RE denoting \( L(x) \cup L(y) \)
    \( \rightarrow \) Any finite list of words can be written as an RE, \( (w_0 \mid w_1 \mid w_2 \mid \ldots \mid w_n) \)
  - \( x^* \) is an RE denoting \( L(x)^* = \bigcup_{0 \leq k < \infty} L(x)^k \)
  - \( x^+ \) is an RE denoting \( L(x)^+ = \bigcup_{1 \leq k < \infty} L(x)^k \)
    \( \rightarrow \) We can use closure to write finite descriptions of infinite, but countable, sets
  \( x^+ \) is just \( xx^* \)

- \( \epsilon \) is an RE denoting the empty set
  \( \rightarrow \) \( \epsilon \) is sometimes useful for writing more concise REs

The operators are concatenation, alternation, and closure.
Regular Expressions

Let the notation $[a...z]$ be shorthand for the RE

$$( \text{a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z} )$$

Examples

Tasteless positive integer $[0...9][0...9]^*$

or $[0...9]^*$

Tasteful positive integer $0 | [1...9][0...9]^*$

Identifier (Algol-like lang) $([a...z]|[A...Z]) ([a...z]|[A...Z]|[0...9])^*$

Decimal number $0 | [1...9][0...9]^* . [0...9]^*$

Real number $((0 | [1...9][0...9]^*) | (0 | [1...9][0...9]^* . [0...9]^*) E [0...9][0...9]^*$
What Is The Point?

What do we care about regular expressions in the context of a compiler?

- We use REs to specify microsyntax — the mapping of spelling to parts of speech
  - An identifier is ( [a...z] | [A...Z] ) ( [a...z] | [A...Z] | [0...9] )*
  - Keywords are specified by their spellings, e.g., if, then, else
  - Those spellings are, in turn, REs

- We use tools derived from automata theory to derive a DFA from the REs and then convert the RE to code that implements a scanner
  - Automatic construction reduces the time & cost of scanner construction
  - Derivation from a formal notation eliminates implementation errors
  - Resulting scanners are both efficient \( O(n) \) and fast (low constant overhead)

- RE-derived scanners are widely used
  - Compilers, text editors, input checking on web pages, software to filter URLs
What Is The Point? RE-derived scanners require $O(1)$ time per character with tiny constant overhead.

Why do we care about regular expressions in the context of a compiler?

- We use REs to specify *microsyntax* — the mapping of spelling to parts of speech
  - An identifier is \(( [a...z] | [A...Z] ) ( [a...z] | [A...Z] | [0...9] )*\)
  - Keywords are specified by their spellings, e.g., *if, then, else*
  - Those spellings are, in turn, REs

- We use tools derived from automata theory to derive a DFA from the REs and then convert the RE to code that implements a scanner
  - Automatic construction reduces the time & cost of scanner construction
  - Derivation from a formal notation eliminates implementation errors
  - Resulting scanners are both *efficient* ($O(n)$) and *fast* (*low constant overhead*)

- RE-derived scanners are widely used
  - Compilers, text editors, input checking on web pages, software to filter *URLs*
A Brief Digression on Time

In COMP 412, we will talk about a lot of “times”

• Design time, build time, compile time, run time, ...

• In practice, the issue of when something happens is one that causes a great deal of confusion among students of compiler construction

  Designs time and build time happen long before compiler runs

  → Costs incurred at design or implementation time do not increase compile time

  Compile time happens every time the user invokes the compiler

  → Users are, appropriately, sensitive to compile time

  → Costs incurred at compile time do not increase run time

  Run-time costs affect actual application performance

  → One critical goal for compilation is to keep run time to a minimum, which means reducing the overhead introduced by translation

As we look at strategies for generating scanners & parsers, keep in mind that generation costs are incurred at implementation time
Automatic Scanner Construction: Meta Issues

**Goals**
- Simplify the construction of robust, efficient scanners
- Develop techniques that have widespread applicability
- Understand the underlying theory & practice

1. Compiler writer creates REs at design time
2. Tools generate the scanner at build time
3. When the compiler runs, it uses the generated scanner to convert source code into a stream of tokens.
Automatic Scanner Construction

Scanner Generator

- May encode its knowledge in tables that drive a “skeleton scanner”
  - Skeleton scanner interprets the tables to simulate the DFA  
    See § 2.5.1

- Every scanner uses the same skeleton, independent of RE

- Scanner generator builds the DFA from the RE, & converts it to a table
Automatic Scanner Construction

Scanner Generator

- May encode its knowledge of the recognizer directly into code
  - Transitions are compiled into conditional logic

- Scanners for different REs are different

- Produces a scanner that has very low overhead per character

- Scanner generator builds the DFA from the RE, & emits code for it

See § 2.5.2
Example from Lecture 2

Recognizer for an ILOC register name (allow redundant zeros)

Rules for DFA Operation

- Start in state $s_0$ & make transitions on each input character
- DFA accepts a word $x$ if and only if $x$ leaves the DFA in an accepting or final state
- If the DFA encounters a character with no specified transition, it moves to $s_e$ & stays in that state
- $r17$ takes it through $s_0, s_1, s_2, s_e$ and it accepts
- $r$ takes it through $s_0, s_1$ and it fails
- ra takes it through $s_0, s_1, s_e$, so it fails

We will use the RE for a register name as a continuing example.

Recognizer for $r$ $[0...9][0...9]^*$

Transitions to $s_e$ are implicit from every state

Any character

ERROR
Example

To be useful, the DFA must be executable

```plaintext
char ← next character
state ← s₀
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← next character
}
if (state is a final state)
    then report success
else report failure
```

Character classifier maps any character into one of the 3 classes: \{r\}, \{0...9\}, \{all others\}

**Skeleton Scanner**

**Transition Table (δ)**

For each character, the skeleton scanner does a table lookup and reads the next character — both of which should be \(O(1)\) operations.

This skeleton scanner is simplified. See Figure 2.14 in § 2.5.1 of EaC2e and the accompanying text.
Example

To capture and classify the lexeme, we add a little work to each state

char ← next character
state ← s₀
lexeme ← null string
while (char ≠ EOF) {
    lexeme ← lexeme || char
    state ← δ[state, char]
    char ← next character
}

If (state is a final state) then {
    category ← f(state)
    return <lexeme, category>
}
else report failure

Transition Table (δ)

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>Any Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
</tbody>
</table>

Skeleton Scanner

Still O(1)
Example

To capture the register number, we would need state-specific actions

```plaintext
char ← next character
state ← s₀
while (char ≠ EOF) {
    state ← δ[state,char]
    char ← next character
    if (state = s₁)
        n ← 0
    else if (state = s₂)
        n ← n * 10 + char – ‘0’
}
If (state is a final state) then {
    category ← f(state)
    return <lexeme,category>
} else report failure
```

Transition Table (δ)

<table>
<thead>
<tr>
<th>state</th>
<th>r</th>
<th>0,1,2,3,4</th>
<th>Any Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
</tbody>
</table>

Skeleton Scanner
Tables

Still O(1)
More Complex REs

What about a more complex language?

• r [0...9] [0...9]* allows arbitrary register numbers \((e.g., \text{r000 or r999})\)
• What if we want to limit the register name to r0 through r31?

Write a tighter specification into the RE

• r ((0|1|2) ([0...9] | \(\varepsilon\)) | (4|5|6|7|8|9) | (3|30|31))
• r0|r1|r2|r3| ... |r31|r00|r01|r02| ... |r09

Each of these REs can be converted to a DFA

• The DFA has the same \(O(1)\) cost per transition
• The DFA takes one transition per input character
• The DFA uses the same skeleton scanner

The added complexity is in the RE, not in the scanner†

† recall the Python documentation
More Complex REs

The DFA for $r \ (0|1|2) ([0...9] \mid \varepsilon) \ (4|5|6|7|8|9) \ (3|30|31) )$

- Accepts a more constrained set of register names
- Still $O(1)$ cost per input character
- More states $\Rightarrow$ more rows in the transition table $\Rightarrow$ more memory
More Complex REs

The DFA for

\[ r \ (0|1|2) ([0...9] \ | \ \varepsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) \ ] \]

- Accepts a more constrained set of register names
- Still \(O(1)\) cost per input character
- More states \(\Rightarrow\) more rows in the transition table \(\Rightarrow\) more memory

Automata Theory Moment
Earlier, we said we would revisit logical complement of an RE or a DFA.
To complement a DFA:
- Make non-final states into final states
- Make final states into non-final states

DFA then accepts any string that the original did not accept \(\Rightarrow\) its complement
This result is not obvious when thinking about the RE.
More Complex REs

The DFA for \( r \ ( (0|1|2) ([0...9] \ | \varepsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) ) \)

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>( r )</th>
<th>0, 1</th>
<th>2</th>
<th>3</th>
<th>4 ...9</th>
<th>Any Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_2 )</td>
<td>( s_2 )</td>
<td>( s_5 )</td>
<td>( s_4 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_e )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_3 \ , \ s_4 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( s_e )</td>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
</tbody>
</table>

Notice that the character classifier has many more divisions than did the earlier one. Still, it should be implementable as a function with \( O(1) \) cost. (see § 2.5)

This table runs in the same skeleton scanner without changes

- To change the language, just change the table
- Still \( O(1) \) cost per character
So far, we have only looked at deterministic automata, or DFAs

- **DFA** ≡ Deterministic Finite Automaton
- Deterministic means that it has only one transition out of a state on a given character

![Diagram of deterministic automaton](image)

*rather than*

![Diagram of non-deterministic automaton](image)
So far, we have only looked at deterministic automata, or DFAs

- DFA \(\equiv\) Deterministic Finite Automaton
- Deterministic means that it has only one transition out of a state on a given character

Can a finite automaton have multiple transitions out of a single state on the same character?
- Yes, we call such an FA a Nondeterministic Finite Automaton
- And, yes, the NFA is one of the more odd notions in CS ... but a useful one

- NFAs and DFAs are equivalent
  - Sometimes, it is easier to build an NFA than to build a DFA

\(\epsilon\)-transition does not consume an input character, which should worry us. \(O(1)\)?
Whoa. What does that **NFA** “mean”?

**Recognizer for an ILOC register name (allow redundant zeros)**

Two Models for NFA Operation:

1. At any nondeterministic choice, the **NFA** clones itself and pursues all choices. If any clone terminates in a final state, the **NFA** accepts.

2. At any nondeterministic choice, the **NFA** makes the correct choice — that it, the one that leads to an accept.

Either model leads to the right intuitions.

And, to break the complexity models, epsilon transitions consume no input. It is okay. We won’t execute **NFAs**

As we used to say in the 1970s, nondeterminism means never having to say you are wrong. *(with apologies to Ali McGraw in “Love Story”)*
Where are we going?

We will show how to construct, for any RE $r$, a deterministic finite-state automaton that recognizes $r$

Overview:

1. Simple and direct construction of a **nondeterministic finite automaton (NFA)**
   to recognize a given RE
   - Easy to build in an algorithmic way
   - Requires transitions on $\varepsilon$ to combine regular subexpressions
2. Construct a **deterministic finite automaton (DFA)** that simulates the NFA
   - Use a set-of-states construction
3. Minimize the number of states in the DFA
   - We will look at 2 different algorithms: Brzozowski & Hopcroft
4. Generate the scanner code
   - Additional specifications needed for the actions
The Plan for Scanner Construction

**RE → NFA** *(Thompson’s construction)*
- Build a *nondeterministic finite automaton (NFA)* for each term in the **RE**
- Combine them in patterns that model the operators

**NFA → DFA** *(Subset construction)*
- Build a **DFA** that simulates the **NFA**

**DFA → Minimal DFA**
- Brzozowski’s algorithm
- Hopcroft’s algorithm

**DFA → RE**
- All pairs, all paths problem
- Union together paths from $s_0$ to a final state

Taken together, these constructions prove that DFAs, NFAs and REs are equivalent.
How Does Class Relate to Regex Libraries?

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. ...

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn’t covered in this document, because it requires that you have a good understanding of the matching engine’s internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

From Python 2.7.10 documentation, emphasis added

- You will learn how to “compile” REs to a DFA & implement a DFA
  - Execution cost is guaranteed $O(1)$ per input character, independent of the expression
- You will have deeper understanding of their power & their use